

University of Canterbury  
Department of Computer Science  
Honours Project Report

Analysis of Algorithms  
Finding the Maximum Flow of a Planar Network

Lynn Loader

Supervisor A. M. Moffat

October 1983

Abstract.

John H. Reif has recently developed an algorithm which finds the minimum cut of a planar network. This is a modification of an earlier algorithm presented by Itai and Shiloach and has a lower theoretical time bound than any other algorithm presented. This report compares the performance and discusses the implementation of Reif's and Itai and Shiloach's algorithms.

## Table of Contents

1	Introduction.....	1
2	Itai and Shiloach's Algorithm.....	6
2.1	The Algorithm.....	6
2.2	To Show that the Algorithm Finds the Minimum Cut.....	10
2.3	Time analysis.....	11
3	Reif's Algorithm.....	12
3.1	Discussion.....	12
3.2	The Algorithm.....	14
4	Implementation.....	16
4.1	Generating Random Graphs.....	16
4.2	Design.....	18
4.3	Data Structures.....	19
4.4	Program Design.....	21
5	Results.....	24
6	Conclusions.....	27

## 1 Introduction.

When a new algorithm to solve a problem is developed, an important part of the discussion of the algorithm is its efficiency with respect to time and space. The measure of the efficiency of an algorithm is often taken as the increase in the time taken for the algorithm to run as the size of the problem increases. This measure is a theoretical time measure and takes no account of the overhead incurred in actually implementing the algorithm ( for example, two algorithms to solve the same problem may have  $O(n^2)$  and  $O(n^3)$  time bounds but the actual time taken to run may be  $200n^2$  and  $2n^3$ , although the  $O(n^2)$  algorithm is theoretically better, in practice it is only better for problems where  $n > 100$  ). Although the theoretical efficiency is important to consider, it is necessary to implement and test the algorithms to find the size problem for which the theoretically better algorithm is actually better, above this level the theoretically better algorithm will always be more efficient.

Many algorithms have been developed to find the maximum flow of a network such as Dinic's [Di] and Karzanov's [Ka] algorithms. Some of these algorithms have been developed to run on a special class of networks known as planar networks. Maximum flow problems turn up often in practice, for example to work out the time to pipe gas between two locations, given the topology of the network, the capacities of the pipe lines and the total amount of gas required.

This report looks at two algorithms for maximum flow in planar networks. Itai and Shiloach [IS] developed an algorithm which runs in  $O(n^2 \log n)$  time. Reif [Re] recently proposed a modification to Itai and Shiloach's algorithm which runs in  $O(n(\log n)^2)$  time, a saving of  $n/\log n$ . The purpose of this report is to describe the implementation of Reif's algorithm and discuss the results obtained from testing, which show that Reif's algorithm is actually better than Itai and Shiloach's on the networks tested.

The rest of this section contains definitions and details of the notation used. Section 2 describes Itai and Shiloach's algorithm, section 3 describes Reif's modifications to this algorithm. Section 4 is concerned with the implementation details of Reif's algorithm, section 5 gives the results of testing carried out on the two algorithms and section 6 contains comments on the results obtained.

#### Definitions.

1. A Graph  $G = (V, E)$  where  $V$  is a set of vertices and  $E$  is a set of edges connecting two vertices  $v_1$  and  $v_2$  in  $V$ . Graphs may be directed, where the edge  $v_1-v_2$  is said to be from  $v_1$  to  $v_2$ , or undirected where the edge may be considered to be an edge in either direction.

2. A Network is a graph in which a vertex  $s$  is called the source vertex and another vertex  $t$  is the sink vertex and a function  $C$  where  $C(e)$  is the capacity ( a non-negative real value ) of the edge  $e$  for each  $e$  in  $E$ .
3. A Flow in a network is a function  $F$  where  $F(e)$  is the flow through the edge  $e$  for each  $e$  in  $E$ .

The flow is subject to two conditions

1.  $0 \leq F(e) \leq C(e)$  for each  $e$  in  $E$ .
2. For all  $v$  a member of  $V - \{s, t\}$  the sum of the flow out of  $v$  equals the sum of that into  $v$ .

The total flow of the network is the sum of the flow out of  $s$  which is the same as the sum of the flow into  $t$ .

4. The Maximum Flow is the flow  $F$  which is maximum while still obeying the flow rules.
5. Planar graphs ( or planar networks ) are graphs ( or networks ) which can be embedded into a plane. If a planar graph has  $n$  vertices then the maximum number of edges it can have is  $3n-6$ . This is derived from Euler's Theorem, which states that the number of vertices plus faces is equal to the number of edges plus 2. See [Ev] for this derivation.

Notation.

Throughout this report the following notation is used.

All logarithms are assumed to be log base 2.

$N$  stands for the number of vertices in a graph.

$K$  stands for the number of vertices on the shortest  $s$ - $t$  path used in both algorithms discussed.

$V$  and  $v(i)$  ( also  $x(i)$  and  $f(i)$  in dual graphs ) are vertices.

$E$  represents an edge.

Ford and Fulkerson's Maximum Flow Minimum Cut Theorem.

Given a network  $N$ , an  $s$ - $t$  cut is a set of edges that if removed from the network, divides it into two disjoint components, one containing  $s$ , the other  $t$ . The number of edges is assumed to be minimal, that is if any edge is removed from the cut, it would no longer be a cut. The value of a cut is the sum of the capacities of all the edges in the cut.

The maximum flow minimum cut theorem states that the maximum flow obtainable in any network is equal to the minimum value taken over all possible  $s$ - $t$  cuts in the network. A proof of this theorem is given in [FF].

Note that while the value of the minimum cut will give the value of the maximum flow, it does not give the flow function  $F$ . Itai and Shiloach [IS] give two algorithms, one of which computes the minimum cut, the other gives the maximum flow function, given the value of this cut. This report concentrates on methods of finding the minimum cut and does not discuss how to obtain the flow function.

## 2 Itai and Shiloach's Algorithm.

Itai and Shiloach [IS] give an algorithm to find the minimum cut of an undirected planar network.

### 2.1 The Algorithm

Given an undirected planar network, the following algorithm will determine the minimum cut.

2.1.1 Find the Dual of the Network. The dual ( $G_d$ ) of a graph ( $G$ ) is a graph related to the original as follows :

1. vertices in  $G$  are faces in  $G_d$ .
2. faces in  $G$  are vertices in  $G_d$ .
3. If an edge from  $v_1$  to  $v_2$  bordering faces  $f_1$  and  $f_2$  exists in  $G$ , there is an edge from  $f_1$  to  $f_2$  bordering faces  $v_1$  and  $v_2$  in  $G_d$ .
4. If  $G$  is a network, the capacity of an edge in  $G$  becomes the cost of the intersecting edge in  $G_d$ .

For a rigorous definition of a dual graph see [Ev].



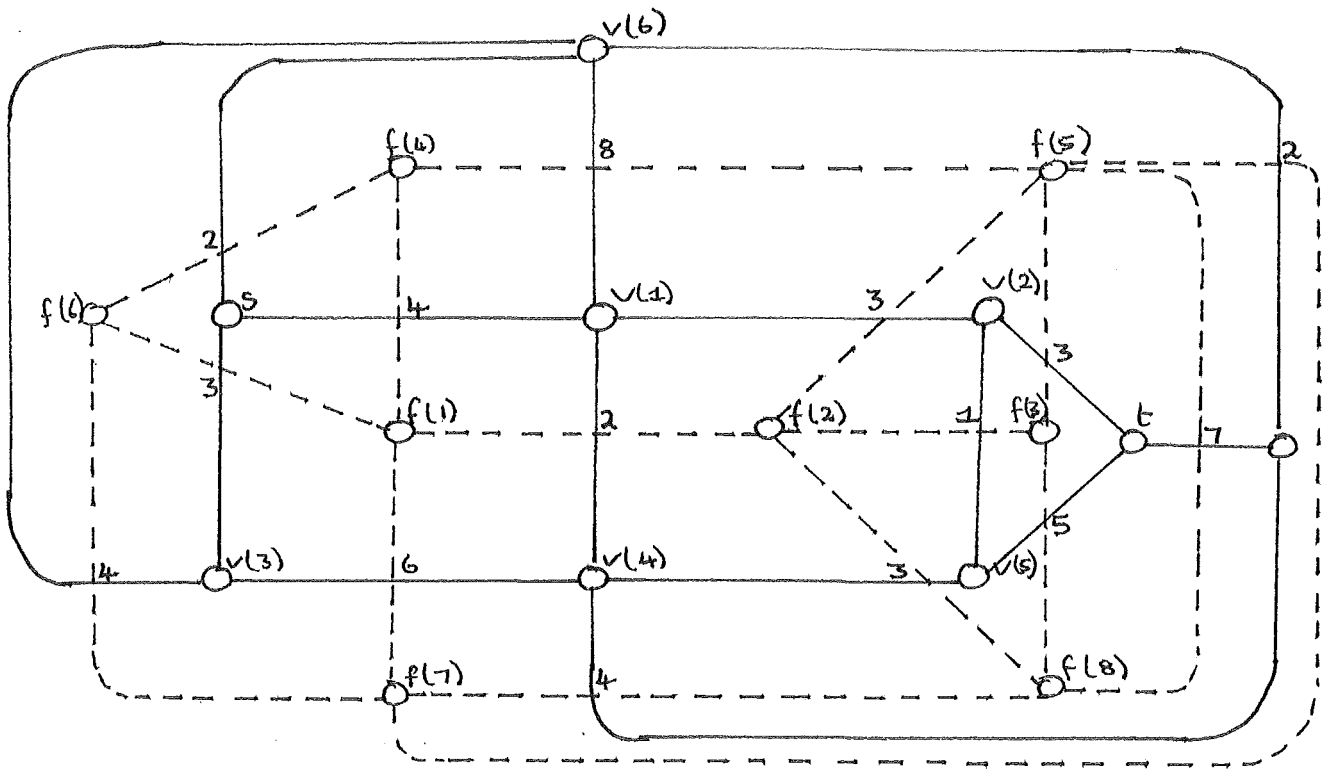


Fig 1. A Graph and its Dual.

2.1.2 Once the dual of the network has been found it is augmented by

1. Adding two vertices,  $s'$  in face  $s$  and  $t'$  in face  $t$ .
2. Adding edges from  $s'$  to every vertex bordering face  $s$  and from  $t'$  to every vertex bordering face  $t$ . All these edges have a cost of 0.

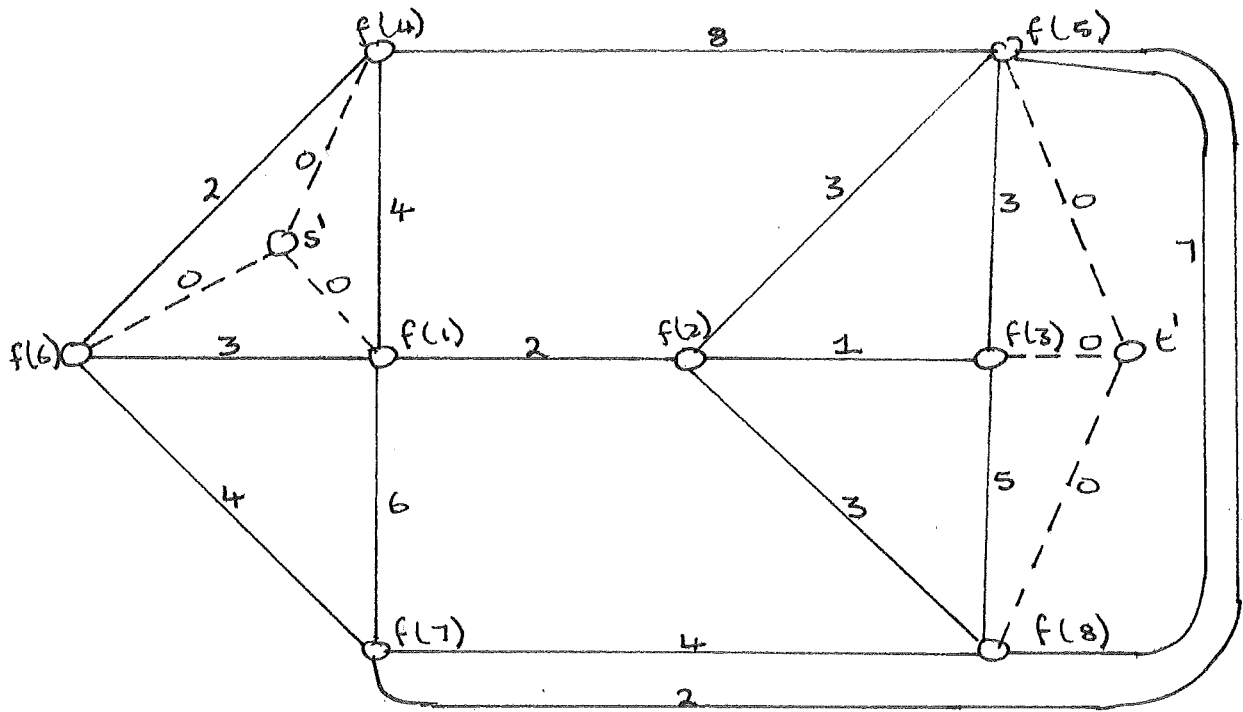


Fig 2. The Augmented Dual.

2.1.3 The Shortest S-T Path. The shortest s-t path of this dual graph is found using Dijkstra's algorithm. The graph is then split along the path as follows :

1. For every vertex  $f(i)$  on this path, ( except  $s'$  and  $t'$  ) a new vertex  $x(i)$  is added and if  $f(i)-f(j)$  is an edge on this path the edge  $x(i)-x(j)$  is added.
2. For each vertex  $f(i)$  on this path and each edge  $e$  connected to  $f(i)$ , except those on the path, if  $e$  is below the path  $e$  is deleted and a new edge between  $f(j)$  and  $x(i)$  is added with the same cost as  $e$ .

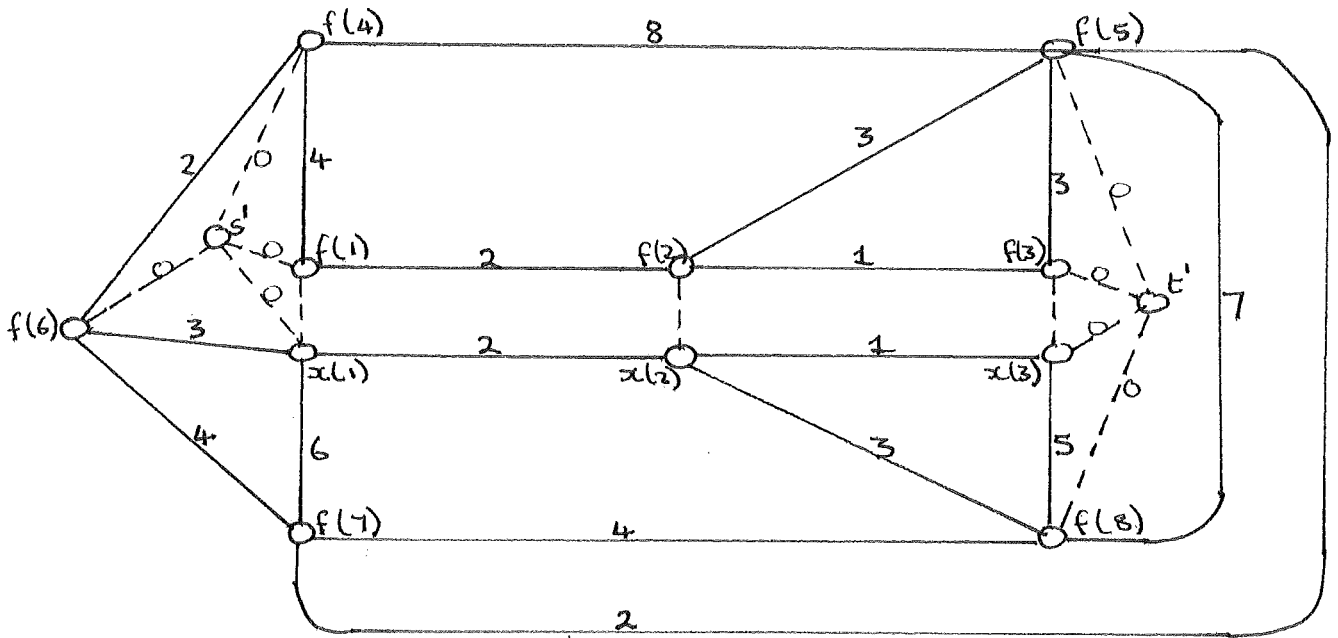


Fig 3. The Dual with the Duplicated S-T Path.

2.1.4 Find Cut Cycles in  $G_d$ . For each pair of vertices  $f(i)$  and  $x(i)$ , the shortest path between them is found and the length of this path is recorded. These paths are the minimum  $f(i)$  cut cycles in the graph. The minimum of these cut cycles corresponds to the minimum cut in the original graph.

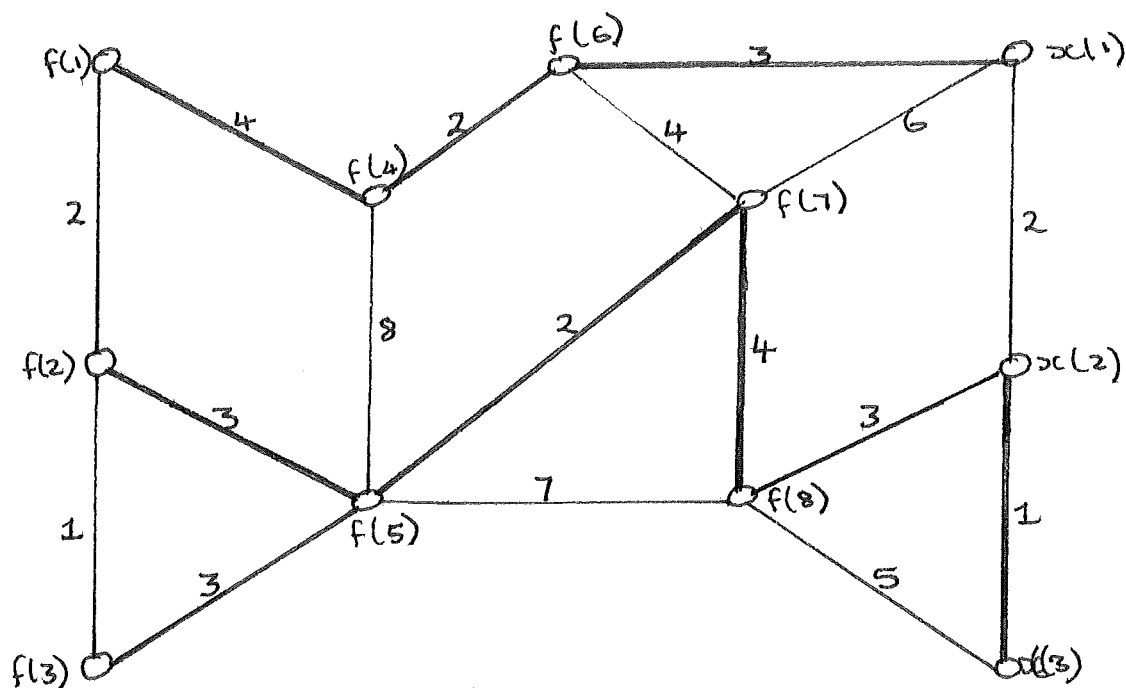


Fig 4. The Dual of Fig 3. with the Minimum  $f(i)$  Cut Cycles.

## 2.2 To Show that the Algorithm Finds the Minimum Cut.

The following two conditions, which hold for all planar undirected networks are necessary to show the minimum cut is found.

1. There is a 1-1 relationship between s-t cuts in  $G$  and the cut cycles in  $G_d$ . The cost of a cycle is the value of the cut.
2. The minimum cut cycle intersects the shortest s-t path and is therefore a minimum  $f(i)$  cut cycle for some  $f(i)$  on this path.

As both the above hold, the minimum s-t cut in  $G$  has a corresponding cut cycle in  $G_d$ . This is the minimum cut cycle in  $G_d$  and so must join the s-t shortest path at  $f(i)$  for some  $i$ . This is the cut cycle found for  $f(i)$  in step 4 of the algorithm and will be the minimum found. It is therefore the minimum cut cycle in  $G_d$  and the algorithm will return

the cost of this cycle as the value of the minimum cut.

### 2.3 Time analysis.

For a graph with  $n$  vertices, the different steps of the algorithm have the following time complexities.

1. To construct dual  $O(m)$ , where  $m = O(n)$  ( ie linear time ).
2. To augment the graph  $O(n)$ .
3. To find the shortest s-t path  $O(n \log n)$

To add the extra vertices and edges  $O(n)$ .

4. To find the shortest paths for each pair  $f(i), x(i)$

Worst case  $O(n^2 \log n)$

Usual case, if there are  $k$  vertices on the shortest s-t path,  $O(kn \log n)$ .

The total complexity of the algorithm is therefore  $O(n^2 \log n)$  in the worst case and  $O(kn \log n)$  in the general case, with step 4 being the dominant contributing factor to this time.

### 3 Reif's Algorithm.

Reif [Re] has developed an algorithm which follows the same structure as Itai and Shiloach. The first three steps are identical, and the modification to the fourth step is described below.

#### 3.1 Discussion.

By modifying step 4 of Itai and Shiloach's Algorithm, Reif produced an algorithm with a general case running time of  $O(n \log k \log n)$  and a worst case time of  $O(n(\log n)^2)$ . His improvement was to notice that if the shortest path between  $f(i)$  and  $x(i)$  was found, the shortest path between  $f(j)$  and  $x(j)$  for  $j \neq i$  would not actually cross it as

1. It must cross in two place ( say at vertices  $v_1$  and  $v_2$  )
2. The shortest path between  $v_1$  and  $v_2$  is found by following the shortest  $f(i), x(i)$  path from  $v_1$  to  $v_2$ .

Therefore any path including vertices  $v_1$  and  $v_2$ , but using another method to get from  $v_1$  to  $v_2$  is not the shortest path and the shortest  $f(j), x(j)$  path does not cross the shortest  $f(i), x(i)$  path, although they may have common edges.

If the middle vertex on the shortest  $s$ - $t$  path is  $f(i)$  and this shortest path is found first, the problem has been divided into two smaller subproblems, the total size of which is  $n$  plus the number of edges on the path just found, as these must be included in both subproblems. If the first subproblem has  $a$  vertices and the second  $b$

vertices, to find the shortest paths for each of the middle vertices of these graphs takes  $O(a \log a)$  and  $O(b \log b)$  respectively.

The total time to find both these paths is  $O(n \log n)$  as

$$\begin{aligned} a \log a + b \log b &\leq a \log(a+b) + b \log(a+b) \\ &\leq (a+b) \log(a+b) \\ &\leq n \log n \end{aligned}$$

This does not take into account the edges common to both graphs, Reif proves that these edges do not increase the bound given above.

If this process is continued until each subproblem has only one shortest path to find, the depth of recursion is  $\log k$ , with each level of recursion taking  $O(n \log n)$  time to compute making the total time to carry out the  $k$  searches  $O(n \log n \log k)$ .

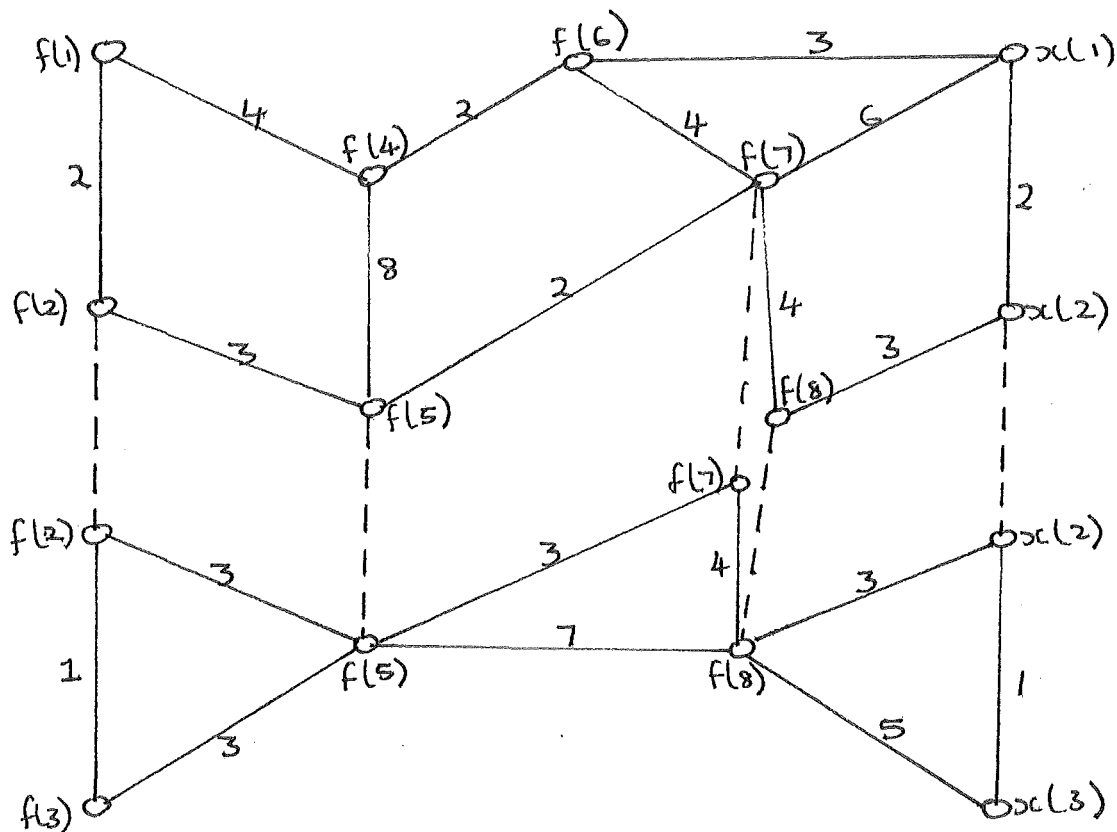


Fig 5. Finding the middle cut cycle splits the problem into two smaller subproblems.

### 3.2 The Algorithm.

The algorithm is a recursive algorithm consisting of the steps

Procedure Calculate Shortest Paths ( upper, lower : vertices ).

1.  $middle := (upper + lower) \bmod 2$ .
2. Find the shortest path from vertex 'middle' to vertex 'middle' + k using only the current subgraph.



3. Mark the shortest path from vertex 'middle' to vertex 'middle' + k as the current lower bound of the graph.

4. If middle <> upper then

    Calculate Shortest Paths ( middle+1, upper ).

5. Turn the lower bound found in step 3 to the current upper bound.

6. If middle <> lower then

    Calculate Shortest Paths ( lower, middle-1 ).

The algorithm is started by Calculate Shortest Paths ( 0, k-1 ), where k is the number of vertices on the s-t shortest path, the vertices 0..k-1 are these k vertices and k..2k-1 are the copies of these vertices. The graph is expected in a slightly modified adjacency list format, as described below.

#### 4 Implementation.

To test the actual running time of Reif's algorithm, the algorithm description in section 3.2 has been implemented. Itai and Shiloach's algorithm for this step has also been implemented as a check to see which is faster. The graphs used for the testing are generated with random edges, but each edge is given a cost of one. This enables the shortest path algorithm to be replaced by a breadth first search and reduces the running time of each algorithm by a factor of  $O(\log n)$  making Reif's run in  $O(n \log k)$  and Itai and Shiloach's  $O(nk)$ . This modification will not affect the comparative running time of the two algorithms. The algorithms are implemented in Pascal on a Prime 750.

##### 4.1 Generating Random Graphs.

For effective testing to be carried out, a method of generating random graphs had to be used. These graphs had to be subject to the conditions that they must be planar and that the shortest path between any two vertices represented as being on the s-t shortest path had to be along that path. The generation of the graph can be divided into three main stages.

4.1.1 Firstly the vertices are generated. The program requires the number of vertices ( $N$ ) in the dual graph and the number of vertices on the shortest  $s$ - $t$  path ( $k$ ) as input.  $K$  vertices ( numbered 0 to  $k-1$  ) are placed evenly along the line  $(0,0)-(0,1)$ , another  $k$  vertices ( numbered  $k$  to  $2k-1$  ) are placed evenly along the line  $(1,0)-(1,1)$ , these represent the vertices on the shortest  $s$ - $t$  path and their duplicate vertices respectively. The other  $N-k$  vertices are given random coordinates in the unit square. The position of each vertex is tested as it is generated to make sure it is not 'close' to any other vertex, close being defined as having a euclidean distance less than  $1/10\sqrt{N}$  in this unit square.

4.1.2 Next the edges in the graph are generated by first constructing the  $s$ - $t$  shortest path by linking the pairs of vertices  $v(i)$  to  $v(i+1)$ , for all  $i$  in the range  $0..k-2$  and  $k..2k-2$ . The other edges are generated by, for each vertex, constructing a list of vertices within a set distance ( required as input data ) from the vertex. This list is sorted according to the euclidean distance between the vertices and the procedure tries to place edges to a random number within the bounds as input, of these vertices. Edges to the vertices closest to the vertex in question are tried first and inserted if they do not disrupt the planarity of the graph. Two edge records are created for each edge and one placed in the list of edges associated with each of the vertices the edge joins, each of which contains a pointer to the other. This is done to make the graph undirected.

4.1.3 The final step is to sort the list of edges associated with each vertex according to the angle the edge makes with the downwards vertical line from the vertex. The linear linked list of edges is then converted to a circular list, ordered anticlockwise.

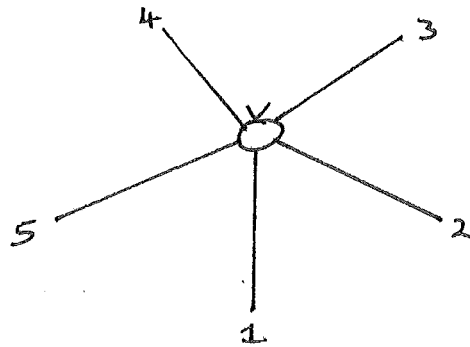


Fig 6. The edges from a vertex are ordered as numbered in this diagram.

See appendix 2 for examples of the graphs generated by these procedures.

#### 4.2 Design.

When designing the implementation of the algorithm, care had to be taken to ensure the running time remained  $O(n \log k)$ . The main effect of this was the need to make the splitting of the graph efficient, with respect to both time and space. Although in Reif's description of the algorithm, he passes the appropriate part of the graph as a parameter to the recursive procedure, this makes a high demand on space for the algorithm, as  $O(\log n)$  copies of the graph are needed. A method of 'screening off' irrelevant parts of the graph was used to ensure only the necessary edges were looked at.

This meant allowing direct access to those edges which form the minimum cut cycles, so the searching algorithm would only consider those edges in the current section of the graph. As a vertex can be on more than one cut cycle, more than one edge may need to be accessed. A list of edges on these cut cycles is stored for each vertex. A new record type is used to store the pointers to the edges concerned. A vertex may also be on both upper bounds and lower bounds at the same time, so two lists need to be kept, one list of upper bounds and one of lower bounds. New bounds are always put on the head of the lower bound lists, so the current lower bound is always the first on the list. The lower bounds must be converted to upper bounds in the reverse order to that in which they were generated, so the bound to be converted will also be the first on the list.

#### 4.3 Data Structures.

The graph is stored as an array of vertices, each one of which points to a circular linked list of edges connected to the vertex. The edges are ordered anticlockwise and the vertex points to the first edge encountered from the downwards vertical line from the vertex. The maximum number of vertices is the constant max\_V which has been set to 3000.

The data structures used are :

Records. The names in brackets give the names of the record types in the Pascal declarations used.

Vertices ( node )

$x, y$  : real. Gives the cartesian coordinates of the vertex  
in the region  $0 \leq x \leq 1, 0 \leq y \leq 1$ .

firste : eptr. Points to the first edge in the list  
associated with the vertex.

father : eptr. Used by the searching procedure and is set  
to point to the edge between the vertex and  
its father vertex in the edge list associated  
with the father vertex.

upper, lower : pathptr. Points to the list of records which  
indicate the edges from this vertex which form  
upper and lower ( respectively ) bounds.

#### Edges ( edgerec )

source, dest : integer. Indicate which pair of vertices the  
edge joins. Source is always equal to the  
vertex associated with the record.

pair : eptr. Points to the equivalent edge in the vertex  
'dest' list.

angle : real. Gives the angle in radians between the line  
 $(G[source].x, G[source].y) - (G[source].x, 0)$   
and the edge.

nexte : eptr. Points to the next edge in the edge list.

#### Paths ( pathrec )

edge : eptr. points to the edge from the vertex on  
this path.

next\_edge : pathptr. Points to the next path record.

next\_V : integer. Gives the next vertex on the path.  
path\_no : integer. Gives the vertex number of the vertex  
the path starts from. This is not used by  
the algorithm but allows the user to  
reconstruct the paths found.

The fields x,y in vertices, angle in edges and path\_no in path are not used by the algorithm and, with the exception of path\_no, are not set by it. They are included for user information and for the ease of getting the graph into the correct format.

The pointers used are EPTRs which point to an edgerec and PATHPTRs which point to a pathrec.

#### 4.4 Program Design.

##### Parameters

G : an array of vertices, each pointing to their edge list  
this is the modified dual of the original graph.

N : The number of vertices in the dual.

k : The number of vertices on the shortest s-t path of the dual.

The total number of vertices expected is thus  $N+k$ . The vertices  $0..k-1$  represent those on the shortest s-t path  $k..2k-1$  represent the copies of these vertices.

The search is implemented as a breadth first search, following the usual algorithm with slight modifications to provide and use extra information needed and obtained by the algorithm.

The queue is a list of edge pointers, this enables the search to record which edge runs between the 'father' vertex and a son, so when a lower bound is set, the appropriate edge can be found directly instead of searching an edge list.

When edges are placed in the queue, instead of placing all the edges associated with the vertex in the queue, a check is made to see if there are any upper or lower bounds associated with the vertex. If there is a lower bound, the first edge to go on the queue is the lower bound, otherwise the edge by which the vertex was reached. Edges are placed in the queue until the upper bound, if one exists, or the starting vertex is reached. Here the list is required to be circular, instead of linear. If the vertex (  $v$  ) is in the range  $k..2k-1$  only one edge is placed on the queue. If the search is started from vertex  $s$  then, if  $k \leq v < s+k$  the edge  $v \rightarrow v+1$  is considered otherwise edge  $v \rightarrow v-1$  is considered. This avoids the need to mark boundaries for edges in this range and also uses the fact that the shortest path from  $v$  to  $s+k$  is  $v, v+1, v+2..s+k$  ( or  $v, v-1, v-2..s+k$  ).

This is where the increased efficiency of Reif's algorithm occurs. As the search is bounded and not complete, the time to perform the search is reduced.



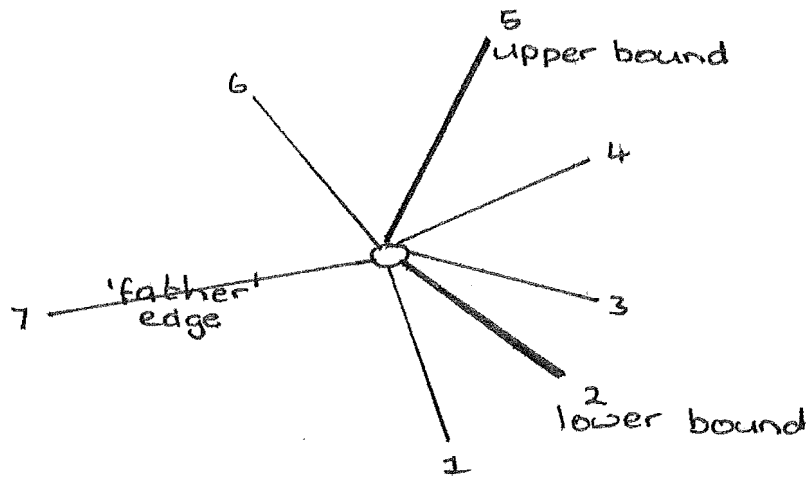


Fig 7. Only the edges between the lower and upper bounds are looked at. Edges 1 and 6 do not need to be considered.

To mark the lower bound, the procedure starts at vertex 'middle+k' and traces back to vertex 'middle', at each vertex ( except middle+k ) a new pathrec is created and set to point to the same edge as the father field of the previous vertex. This procedure also records the length of the path.

To turn a lower bound into an upper bound, a procedure is called to trace along the lower bound starting from vertex 'middle', remove the first record from the list of lower bounds and place it at the head of the list of upper bounds.

## 5 Results.

The following graph shows the results of tests run with  $N = 2^x$ ,  $x = 2, 3 \dots 8$  and  $k = \sqrt{n}$ . The scales are logarithmic on both axes.

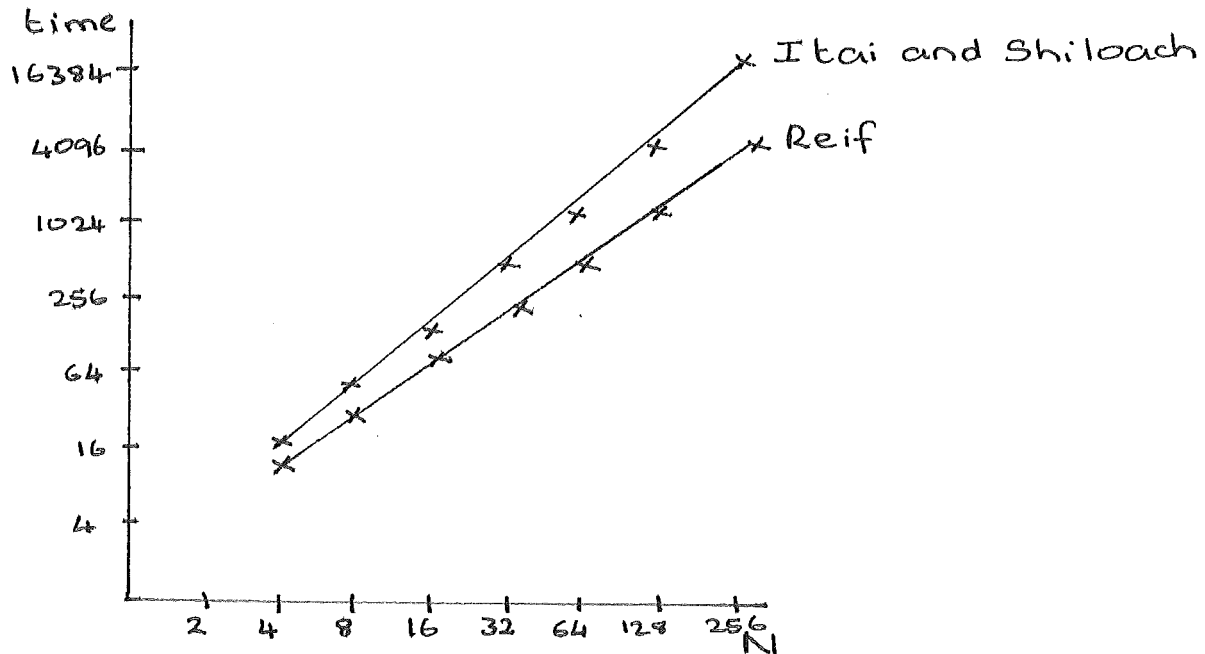


Fig 8. Graph of times ( in milliseconds ) obtained by running Reif's and Itai and Shiloach's algorithms.

The table shows the mean values found for the different  $N$ . As can be seen, Itai and Shiloach's algorithm produces far higher times than Reif's, running over twice as fast for  $N > 16$ . Even below this, Reif's algorithm is still good.

N	k	Reif	Itai and Shiloach	$\frac{\text{Reif}}{n \log k}$	$\frac{\text{I\&S}}{n \cdot k}$
4	2	17	21	4.25	2.63
8	3	36	60	2.84	2.50
16	4	83	152	2.59	2.38
32	6	231	525	2.79	2.73
64	8	501	1388	2.61	2.71
128	11	1275	4996	2.88	3.55
256	16	3164	16745	3.09	4.09

Table 1. Times ( in milliseconds ) obtained from runs of Reif's and Itai and Shiloach's algorithms, varying N.

*values  
for graph*

N	k	Reif	Itai and Shiloach	$\frac{\text{Reif}}{n \log k}$	$\frac{\text{I\&S}}{n \cdot k}$
256	4	2156	4177	4.21	4.08
256	8	2806	8466	3.65	4.13
256	16	3164	16745	3.09	4.09
256	32	4106	35373	3.21	4.32
256	64	4979	72109	3.24	4.40

Table 2. Times obtained by running the algorithms, varying k.

The last two columns of the tables give an indication of how close the times obtained are to the theoretical time bounds. For most pairs of N and k these are reasonably constant, indicating that the times are close to those expected.

These figures indicate that the overhead incurred through implementing Reif's algorithm is not high enough to make it infeasible to use in 'real' situations.

## 6 Conclusions.

As can be seen from the results obtained, Reif's algorithm ran consistently faster than Itai and Shiloach's. The algorithm is more complicated than Itai and Shiloach and hence requires more programming effort, but once done achieves an elegant implementation. The data structure used is also more complicated. The overhead incurred by running this implementation and maintaining the data structure is surprisingly low, yielding good results on all the types of graphs tested.

This implementation shows that Reif's algorithm is a more efficient algorithm in practice as well as theoretically and so is a useful method of finding the minimum cut of a planar network. The implementation is more elegant than expected because of the method used to turn lower bounds into upper bounds. This turned out to be easier than originally anticipated.

### Further Testing.

The following areas, which have not been implemented or tested, could be used for a more complete analysis of Reif's algorithm.

6.1 Generate graphs with edges having random capacity, and change the searching technique from a breadth first search to Dijkstra's shortest path algorithm, as would be needed in any practical implementation of the algorithm.

6.2 Implement the full algorithm - finding and augmenting the dual given the original graph, and finding the shortest s-t path in the dual - as well as the implementation described here. This, together with an algorithm to find the flow function, gives an algorithm to find the maximum flow function. Testing this with other general algorithms such as Karzanov's [Ka] will show which is better overall, as the overhead incurred by Reif's algorithm may be high compared to that of Karzanov's  $O(n^3)$  algorithm.

References.

[DI] Dinic E. A.

Algorithm for Solution of a Problem of Maximal Flow in a  
Network with Power Estimation.

Soviet Math. Dokl Vol 11 ( 1970 ) pp 1277-1280.

[Ev] Even S.

Graph Algorithms.

Pitman, London, 1979.

[FF] Ford C. and Fulkerson D.

Maximal Flow through a Network.

Canadian Journal of Mathematics Vol 8 ( 1956 ) pp 399-404.

[IS] Itai Alon and Shiloach Yossi.

Maximum Flow in Planar Networks.

Siam Journal of Computing Vol 8 ( 1979 ) pp 135-150.

[Ka] Karzanov A. V.

Determining the Maximum Flow in a Network by the  
Method of Preflows.

Soviet Math. Dokl Vol 15 ( 1974 ) pp 434-437.

[Re] Reif John H.

Minimum S-T Cut of a Planar Undirected Network  
in  $O(n \log^2 n)$  Time.

Siam Journal of Computing Vol 12 ( 1983 ) pp 71-81.

Appendix 1. Program Listings  
 1.a Code for Reif's Algorithm.

```

subprogram reif;

Procedure search_graph ( var G : graph; v, N, k : vertex;
{
    ===== } var visited : integer );

{ This procedure performs a breadth first search on
graph G, starting from vertex v. }

type
  listptr = ^listmember;
  listmember = record
    edge : eptr;
    next : listptr
  end ;

var
  labelled : array [vertex] of boolean;
  head : listptr;
  edge, place, stop : eptr;
  i : integer;

  Procedure queue ( edge : eptr );
  {
    =====

    Places 'edge' into the queue of edges used by the search. }

  var temp, trace : listptr;

  begin
    visited := visited + 1;
    new ( temp );
    temp^.next := nil;
    temp^.edge := edge;
    trace := head;
    if trace <> nil then
      begin
        while trace^.next <> nil do trace := trace^.next;
        trace^.next := temp
      end
    else head := temp
  end ; { Queue }

  Function fetch : eptr;
  {
    =====

    Fetches the first edge from the queue and returns
    a pointer to the edge. }

  var temp : listptr;

  begin

```



```

fetch := head^.edge;
temp := head;
head := head^.next;
dispose ( temp )
end ; { fetch }

```

```

Procedure Set_limits ( var start, stop : eptr; v1 : vertex );
{
    =====

```

This procedure sets the limits on the edges to be added to the queue maintained by the searching procedure. All the edges from START up to ( but not including ) STOP are to be added to the queue. If there is a lower bound from the vertex, START is set to this, otherwise START is set to the current edge. STOP is set to the edge following the upper bound, if it exists, or to the current edge. }

```

begin
with G[v1] do
    if ( v1 >= k ) and ( v1 < 2*k ) then
        begin
            if ( v1 = k ) or ( v1 >= v+k ) then start := firste
            else start := firste^.nexte;
            stop := start^.nexte
            end
        else
            begin
                if upper = nil then stop := edge^.pair
                else { get the upper bound }
                    stop := upper^.edge^.nexte;
                if lower = nil then start := edge^.pair
                else { get the lower bound }
                    start := lower^.edge;
                end
            end
        end ; { set_limits }

```

```

begin
for i := 0 to N+k-1 do labelled[i] := false;
visited := 0;
new ( head );
head^.next := nil;
head^.edge := G[v].firste^.pair;
while head <> nil do
    begin
        edge := fetch;
        with edge^ do
            if not labelled[dest] then
                begin
                    labelled[dest] := true;
                    G[dest].father := edge;
                    set_limits ( place, stop, dest );
                    repeat
                        queue ( place );
                        place := place^.nexte

```

```

        until place = stop;
        end
    end;
    G[v].father := nil
end ; { search graph }

```

```

Procedure Minimum_cut ( var G : graph; N, k : vertex );
{
    =====

```

This procedure finds the value of the minimum cut of a planar undirected graph using Reif's algorithm. It expects the dual of the graph for parameter G, with N vertices and k vertices on the shortest s-t path. It also assumes the graph has been split along this path. }

```

var
    i, length, min_cut : integer;
    path_length : array [vertex] of integer;
    visited : array [vertex] of integer;
    time : array [vertex] of integer;
    t1, t2 : integer;

```

```

Procedure lowertoupper ( v : vertex );
{
    =====

```

Converts the path from vertex v to vertex v+k from being a lower bound to being the current upper bound by changing the pathrec records at the head of the 'lower' lists to being the head of the 'upper' lists. }

```

var
    i : integer;
    temp : pathptr;

```

```

begin
    i := v;
    while i <> v + k do
        with G[i] do
            begin
                temp := lower;
                lower := temp^.next_edge;
                temp^.next_edge := upper;
                upper := temp;
                i := upper^.next_v
            end;
        end; { lower to upper }
    end;

```

```

Procedure marklower ( v : vertex; var length : integer );
{
    =====

```

Puts the path from vertex v to v+k as the current lower bound on the graph by placing new pathrec records at the head of the 'lower' lists. It also calculates the length of the path, which is returned in parameter length. }

```

var
  i, j : integer;
  temp : pathptr;

begin
  i := v + k;
  if G[i].father = nil then halt ( 'Graph not connected' );
  length := 0;
  while i <> v do
    begin
      j := G[i].father^.source;
      new ( temp );
      with temp^ do
        begin
          next_v := i;
          path_no := v;
          edge := G[i].father;
          next_edge := G[j].lower;
          G[j].lower := temp;
        end;
      i := j;
      length := length + 1
    end;
  end ; { mark lower }

  Procedure calculate ( lower, upper : integer );
  {
    =====

    Recursively calculates the shortest path from i to
    i+k for all lower <= i <= upper, using a 'divide and conquer'
    technique. The values of all these shortest paths are stored
    the array path_length.
  }

  var
    middle : vertex;
    t1, t2 : integer;

  begin
    mill ( t1 );
    middle := ( upper + lower ) div 2;
    G[middle+k].father := nil;
    search_graph ( G, middle, N, k, visited[middle] );
    marklower ( middle, path_length[middle] );
    if middle < upper then calculate ( middle+1, upper );
    lowertoupper ( middle );
    if lower < middle then calculate ( lower, middle-1 );
    mill ( t2 );
    time[middle] := t2 - t1;
  end; { calculate }

begin { minimum cut }
mill ( t1 );
for i := 0 to N+k-1 do
  begin

```

```

    G[i].upper := nil;
    G[i].lower := nil
  end;
calculate ( 0, k-1 );
mill ( t2 );
writeln ( 'Vertex Pathlength Edges searched Time' );
writeln ( '====' );
min_cut := path_length[0];
writeln ( 0:4, path_length[0]:10, visited[0]:12, time[0]:11);
for i := 1 to k-1 do
  begin
    writeln ( i:4, path_length[i]:10, visited[i]:12, time[i]:11 );
    if path_length[i] < min_cut then min_cut := path_length[i];
  end;
writeln ( 'The value of the minimum cut is ', min_cut : -4 );
writeln ( 'The total search time was ', t2-t1:-8 );
end ; { minimum cut }

```

1.b Code for Itai and Shiloach's Algorithm.

subprogram breadth;

Procedure bfs ( var G : graph; v, N : vertex );  
{  
    ===

    This procedure performs a breadth first search on  
graph G, starting from vertex v. }

type  
    listptr = ^listmember;  
    listmember = record  
        edge : eptr;  
        next : listptr  
    end ;

var  
    labelled : array [vertex] of boolean;  
    head : listptr;  
    edge, place, stop : eptr;  
    i : integer;

Procedure queue ( edge : eptr );  
{  
    =====

    Places the edge source to dest into the queue of  
edges used by the search. }

var temp, trace : listptr;  
begin  
    new ( temp );  
    temp^.next := nil;  
    temp^.edge := edge;  
    trace := head;  
    if trace <> nil then  
        begin  
            while trace^.next <> nil do trace := trace^.next;  
            trace^.next := temp  
        end  
    else head := temp  
end ; (\* Queue \*)

Function fetch : eptr;  
{  
    =====

    Fetches the first edge from the queue and returns  
a pointer to the edge. }

var temp : listptr;

begin  
    fetch := head^.edge;  
    temp := head;

```

head := head^.next;
dispose ( temp )
end ; (* fetch *)

Procedure Set_limits ( var start, stop : eptr;
                      dest : vertex );

begin
start := G[dest].firste;
stop := start
end ; { set_limits }

begin
for i := 0 to N do labelled[i] := false;
new ( head );
head^.next := nil;
head^.edge := G[v].firste^.pair;
while head <> nil do
begin
edge := fetch;
with edge^ do
if not labelled[dest] then
begin
labelled[dest] := true;
G[dest].father := edge;
set_limits ( place, stop, dest );
repeat
queue ( place );
place := place^.nexte
until place = stop;
end
end;
G[v].father := nil
end ; (* bfs *)

Procedure Check_search ( var G : graph; N, k : vertex );
{
=====

Runs Itai and Shiloach's minimum cut algorithm on
graph G and outputs the k shortest paths and the time
taken to run the procedure.
}

var i, j, count, t1, t2 : integer;
begin
mill ( t1 );
for i := 0 to k-1 do
begin
bfs ( G, i, N+k );
count := 0;
j := i + k;
while j <> i do
begin
count := count + 1;
j := G[j].father^.source;

```

```
        end;  
        writeln ( i : 5, count : 5 );  
    end;  
mill ( t2 );  
writeln ( 'Time to check search = ', t2-t1 : 5 );  
end; { check search }
```

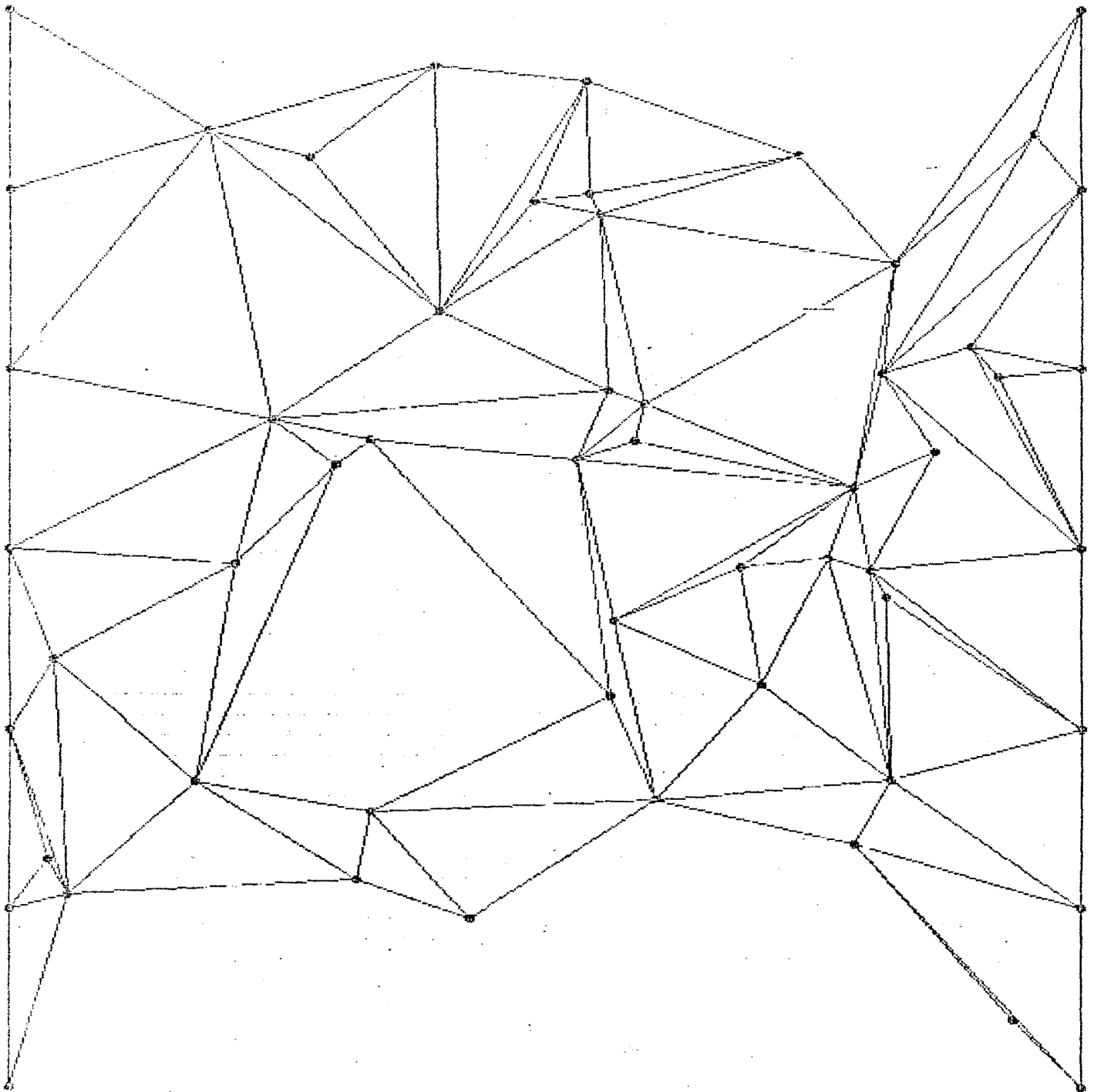
## 1.c Data Structure Declarations.

```
const
  max_V = 3000; { max number of vertices }
  max_E = 9000; { max number of edges ( 3*max_V ) }

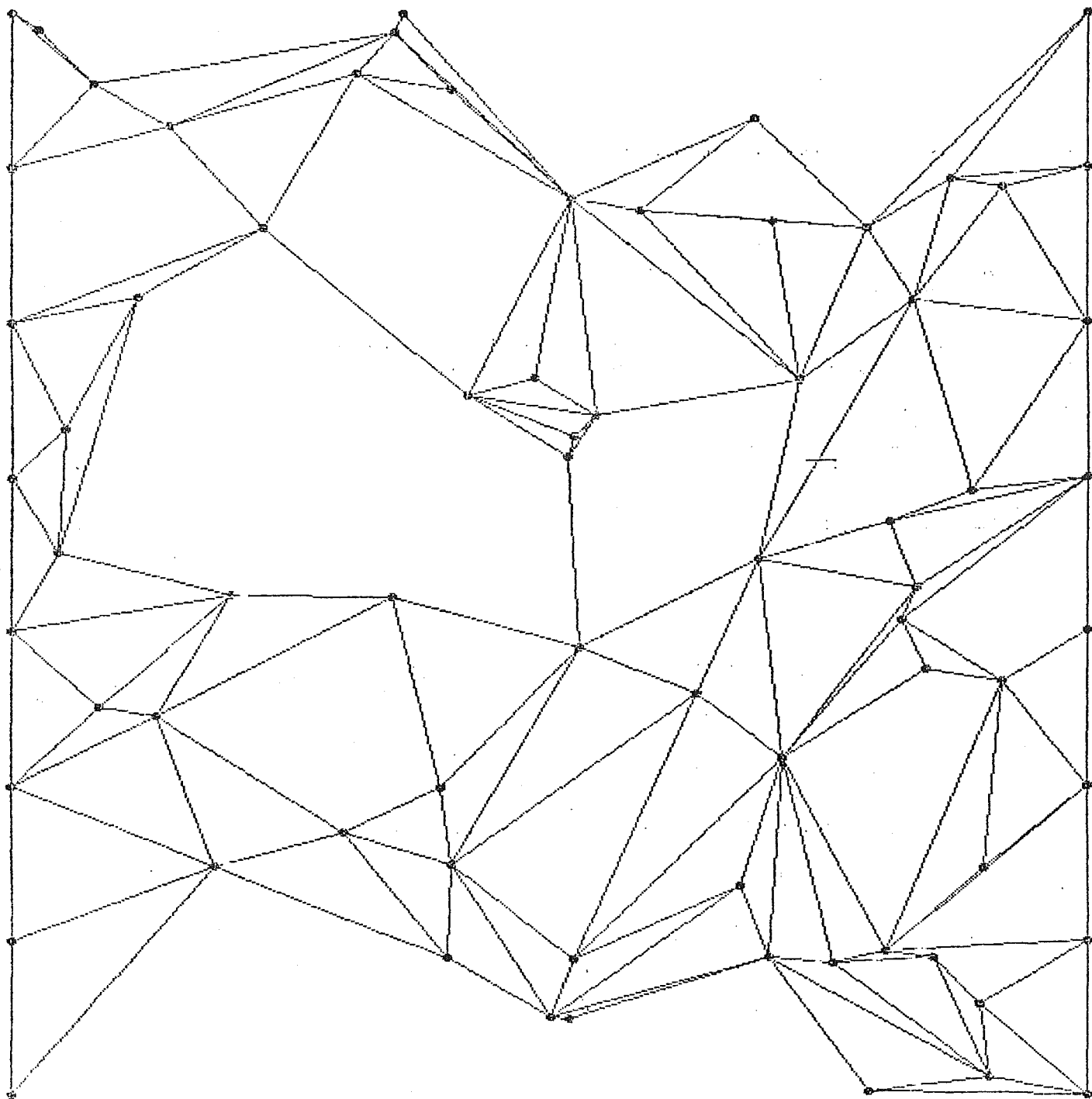
type
  vertex    = 0..max_V; { numbers for vertices }
  eptr      = ^edgerecs;
  edgerecs  = record { an edge }
    source  : integer;
    dest    : integer;
    angle   : real;
    pair    : eptr;
    nexte   : eptr
  end;
  pathptr   = ^pathrec;
  pathrec   = record { marks an edge boundary }
    next_v   : integer;
    path_no  : integer;
    edge     : eptr;
    next_edge : pathptr
  end;
  nodes     = record { a vertex }
    x, y     : real;
    father   : eptr;
    firste   : eptr;
    upper    : pathptr;
    lower    : pathptr
  end;
  graph     = array [vertex] of nodes;
```



Appendix 2. Examples of Random Graphs Generated for Testing  
2.a  $N = 49$ ,  $k = 7$



2.b  $N = 64, k = 8$



2.c  $N = 128, k = 11$

